ISBN 978-83-60810-22-4

Computer Language Notation Specification through Program Examples

Miroslav Sabo, Jaroslav Porubän, Dominik Lakatoš, Michaela Kreutzová
Technical University of Košice
Letná 9
042 00 Košice, Slovakia
Email: {miroslav.sabo, jaroslav.poruban, dominik.lakatos, michaela.kreutzova}@tuke.sk

Abstract—**It often happens that computer-generated documents originally intended for human recipient need to be processed in an automated manner. The problem occurs if analyzer does not exist and therefore must be created ad hoc. To avoid the repetitive manual implementation of parsers for different formats of processed documents, we propose a method for specification of computer language notation by providing program examples. The main goal is to facilitate the process of computer language development by automating the specification of a notation for recurring well-known language constructs often observed in various computer languages. Hence, we introduce the concept of language patterns, which capture the knowledge of language engineer and enables its automated application in the process of notation recognition. As a result, by using the proposed method, even a user less experienced in the field of computer language construction is able to create a language parser.**

I. Introduction

SOFTWARE systems generate a lot of textual output, either as a main product of their execution or for other secondary purposes such as logging. If the output is intended for information transfer and further processing by another system, its structure must be explicitly defined (e.g. XML and XSD) in order for receiving system to transform the textual content into appropriate structural representation. On the other hand, if the output is intended for human recipient, the structure of textual content is often not defined explicitly but is rather hidden in its human-usable notation. In most of these cases, the explicit specification of structure is not even necessary, as primary purpose of such textual output is to store the information in form which is easily comprehensible to human users just by reading it. However, sometimes it happens that output originally intended for human has to be processed by computer (e.g. to perform an analysis). Since explicit specification of its structure does not exist, it must be defined first.

Considering the generative origin and human-usable notation, we think of such textual output as of collection of programs written in some domain-specific language (DSL). From this perspective, the task of specifying a structure can be translated into a *problem of DSL notation specification*. One of the options is to examine the source code of the system generating the output and construct the grammar of a DSL accordingly. Besides the complexity of such task, it requires access to source code, which might not always be feasible. That leaves us the specification of a notation by inferring it from the provided program examples.

Although syntax recognition is a well-established area of research and multiple approaches to grammar inference have already been implemented [1]–[3], in this paper we propose a method for example-driven DSL notation specification (EDNS) based on language patterns. The innovative concept of *language patterns* is proposed to capture the well-known recurring notation of common language constructs often seen in many computer languages. Automated identification of such patterns in provided program examples eliminates the necessity to specify the same notation manually and repeatedly for each language being designed. Along the recognition of recurring notations, language patterns are also used to check whether the notation satisfies the conditions (e.g. unambiguity) of a machine-processable computer language.

Our objective is to automate the process of DSL notation specification as much as possible. By using EDNS method, it is be possible to perform this task even for a user with little knowledge about the language construction. Moreover, the concept of formalized language patterns allows to incorporate into the process a domain expert with no technical knowledge as well. Besides the analytic way of using language patterns in context of EDNS method, they can also be utilized in the synthetic manner when specifying a notation by their composition [4].

The rest of the paper is organized as follows. In the next section we give a detailed description of approaches to computer language inference, with special emphasis on DSLs. The overview of EDNS method is given in Section III. Following sections discuss the individual artifacts used in the method – Section IV describes the specification of abstract syntax of a DSL, while Section V describes ProgXample, the mechanism for formalization of textual program examples which represent the concrete syntax of a DSL. Section VI elaborates in detail on the proposed concept of language patterns and discusses its application in the proposed EDNS method. Finally, Section VII gives the conclusions of the paper.

II. Related Works

The problem of syntax recognition from existing resources has been specifically addressed by grammar inference research community. Successful results have been achieved in inferring regular languages using various algorithms like EDSM [5] and RPNI [6]. Genetic-programming approach was used for inferring regular grammars in [2]. The methodologies of context-free grammars induction for domain-specific languages are the aim of GenParse Project research group. Their initial research of genetic approach resulted in the successful induction of small grammars [7], however their current focus is on the incremental grammar learning [1] which should allow for bigger DSLs to be inferred as well.

Although yet another various approaches both automatic [8] or semi-automatic [9], [10] exist, they are all targeted towards language design based on the concrete syntax which is recognized solely from artifacts represented as sentences/programs written in the unknown language. The aim of our research is focused, however, on the language development based on abstract syntax since we advocate that language specification with separated abstract and concrete syntaxes is more suitable for DSL design. The fact that many supporting tools for DSL development [11], [12] follow this approach makes for a promising research area to explore.

III. Example-Driven Method for DSL Notation Specification

In traditional approach to computer language development [13], central part of the language specification is grammar which defines the concrete syntax (notation) of a language. Abstract syntax is not defined explicitly but can be derived from the grammar.

In model-driven approach to DSL development, it is a usual practice to define abstract and concrete syntaxes separately [14], [15], using some metamodeling and grammar-like languages respectively. The DSL notation is specified formally by writing grammar-like rules containing domain-specific keywords and references to elements of language metamodel. This process is performed by language engineer. As the required domain-specific notation is achieved by amending the rules with domain-specific keywords, we can look at the process of notation specification as a tailoring of the computer-like syntax to meet the requirements on look and feel of the specific domain.

Example-driven method for DSL notation specification (EDNS) takes the opposite direction and starts with ideal domain-specific notation, provided informally by domain expert through examples of programs as they would have been written in a desired DSL (Fig. 1). The formal specification is then derived from examples using the concept of language patterns. The method consists of three consecutive phases – *formalization of program examples*, *DSL notation recognition* and *generation of language specifications*.

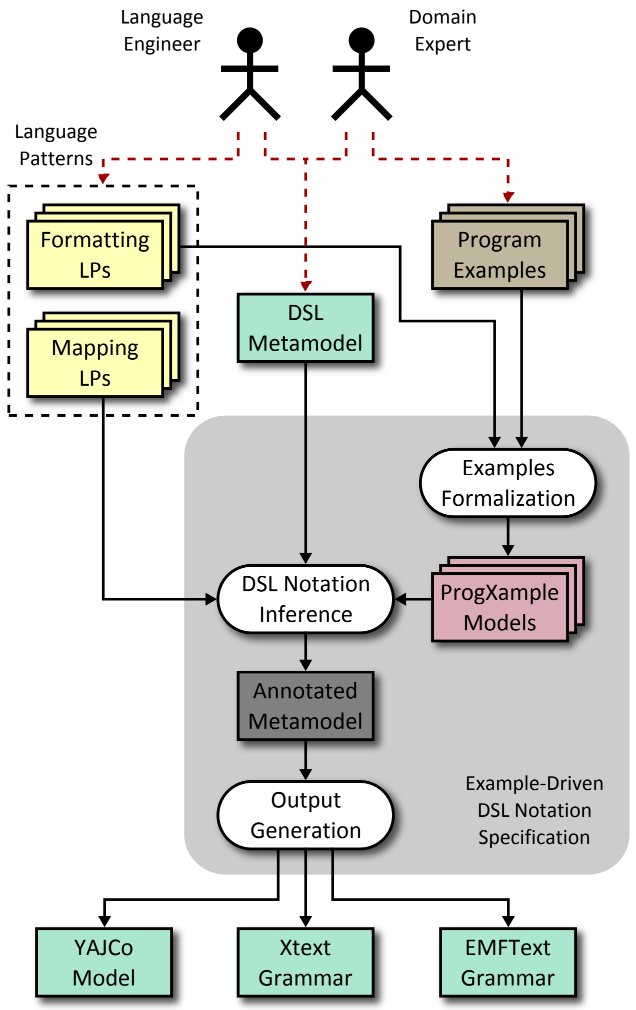

Fig. 1. Overview of the example-driven method for DSL notation specification.

A. Formalization of Program Examples

Formalization of Program Examples can be considered a preprocessing phase to the main phase of EDNS method where the actual process of notation recognition happens. Its purpose is to formalize the program examples which are given informally as plain text files, so they can be processed in an automated manner in the following phase.

The resulting formal representation is tree models defined in Ecore-based *ProgXample modeling language*. The process of formalization is automated and models are created by transforming the initial models using the *formatting language patterns*. Initial models are constructed as trivial trees with single node holding the whole textual content of an example file. Creating of ProgXample models will be discussed in more detail in Sec. V.

B. DSL Notation Inference

DSL Notation Inference is the main phase of EDNS method. It has three inputs – *DSL metamodel*, *ProgXample models* and *mapping language patterns*. DSL metamodel represents the abstract syntax of a language and is the result of common domain analysis conducted by both language engineer and domain expert. ProgXample models are the output of previous preprocessing phase and they are the formalization of concrete syntax proposed by domain expert. The last input is a set of language patterns which drive the process of notation recognition. Patterns are in an iterative manner compared against metamodel and ProgXample models and if the match is found, the appropriate elements of metamodel are injected with special annotations. The output of the notation recognition process is the metamodel enriched with annotations marking the identified patterns. This metamodel is an internal representation of the language specification in EDNS method.

C. Output Generation

EDNS method does not directly concern the implementation of languages or their supporting tools. For this purpose, it rather exploits the maturity of existing technologies in this area. There are many tools for language design and implementation, varying from simple parser generators [4] to complex language workbenches [14], [15], which can generate the full implementation of a language given just its specification. Moreover, the more sophisticated tools can generate other supporting tools (editor, debugger, visualizer, etc.) or even a fully-fledged language-specific IDE as well. These tools are in the focus of EDNS method and to utilize their generative capabilities, EDNS provides them with language specifications in the appropriate format.

Output generation is the final phase of EDNS method. It takes a single input produced by the preceding phase of language notation recognition – annotated metamodel of a language. At first, Ecore metamodel serves as a source for instantiating the in-memory Java objects which constitute the internal YAJCo model [16]. This model represents the implementation-independent language specification and itself can be used for generation of the language parser and supporting language-specific tools. The specification files for other tools for language implementation (e.g. Xtext, EMFText) can be generated from YAJCo model using the Generator tool provided by YAJCo framework [16].

IV. Language Metamodel

Language metamodel is the central part of computer language specification in the proposed EDNS method, as it defines the abstract syntax of a language. It is composed of domain concepts, their properties and relations between the concepts. The metamodel is created to formally capture the output of domain analysis conducted together by both language engineer and domain expert. In EDNS method, a metamodel is constructed using the *EMF Ecore Metamodel*. Each domain concept is represented by a single class and concept's properties are modeled using the class attributes. The relations between concepts are in metamodel represented as named connections between classes. Since metamodel is defined in EMF Ecore format, any of the existing EMF editors can be utilized in this phase.

V. ProgXample Models

In EDNS method, concrete syntax of a language is defined informally by providing the examples of programs as they would have been written in a DSL being specified. For these examples to be processable in the following phase of automated notation recognition, first they must be formalized. For this purpose, a specialized modeling language, tailored for the needs of EDNS method, has been proposed.

ProgXample is a compositional modeling language for tree representation of textual content. It is defined on EMF Ecore platform by its metamodel which specifies the components of tree models that can formally represent the plain text of program examples. The special facet of this language is that its metamodel is not defined as a monolithic structure, but it is rather composed of extensions attached to the core metamodel.

The construction of trivial tree is the starting point of formalization of every program example and it is same for every example file. Comparing the trivial tree model to a flat representation of textual content, it indeed does not bring any additional information on the structure of the program example. However, tree model is iteratively compared against formatting language patterns and if the match is recognized, the model gets transformed into more complex form, possibly being augmented with nodes of other kinds, introduced to metamodel by various patterns. After all patterns have been compared, the final form of tree models is handed as an input to the following phase of DSL notation recognition.

VI. Language Patterns

The concept of *language pattern* in the area of computer language design has been inspired by the concept of design pattern in the area of object-oriented software design [17]. In analogy with design patterns, language patterns capture the language design knowledge in a form that can be reused effectively. The captured knowledge serves two purposes:

1) captures the widely known and accepted notation of particular language constructs, often used throughout various programming languages (e.g. punctuation, delimiter marks, bracketing)
2) controls the specified notation that it satisfies requirements on being computer-processable (e.g. problem of ambiguity in a language)

Language patterns are the foundation of a proposed method for example-driven DSL notation specification [18]–[20]. According to their utilization, they come in two flavors – *formatting language patterns* and *mapping language patterns*.

Formatting language patterns (FLPs) are used in the early phase of EDNS method when ProgXample models are being created. Their purpose is to formalize the plain text of example files into form that will be processable in the following phase of notation inference. FLPs only concern the concrete syntax of a language therefore the only artifacts included in the process of pattern recognition are program examples which define the notation of a language.

Mapping language patterns (MLPs) are used in the main phase of EDNS method when the actual process of DSL notation inference happens. Their purpose is to infer the concrete human-usable notation of a language from the provided program examples. The inferred notation is then defined as mapping between abstract and concrete syntaxes. Since MLPs concern both syntaxes of a language, the artifacts involved in the process of inference include both DSL metamodel representing the abstract syntax and ProgXample models (formalized program examples) representing the concrete syntax.

VII. Conclusion

Capturing the recurring notation style of common language constructs and its formalization in form of computer language patterns is an unexplored topic in the area of computer language design. In this paper we have elaborated on this novel idea and have discussed its application in context of model-driven language development. The proposed method for example-driven DSL notation specification (EDNS) has been introduced. The paper has presented in detail the concept of formatting and mapping language patterns and its application in EDNS method.

The language patterns open new possibilities in construction of computer languages. They can be utilized in both directions to creating a language specification, analytical and synthetic. Although synthetic approach was not discussed in this paper, more details on this subject can be found in our earlier work [4]. This paper presented the analytical approach, set up in the context of proposed EDNS method. We believe that by using it there is a potential for speeding up the process of language creation since DSL notation is inferred from program examples automatically and the only part of DSL specification that must be performed manually by language engineer boils down to construction of DSL metamodel. Besides the cases where program examples are provided as an output generated by software, the examples can be created manually as well. This gives a possibility to define DSL notation even for a person not versed in language construction, nevertheless the most competent for such task – a domain expert. Since having a domain expert as the direct author of DSL notation can significantly increase the quality and usability of developed DSL, we consider this an important benefit offered by EDNS method.

Currently we are working on development of graphical user interface for EDNS method. This will enable the visualization of identified language patterns in provided program examples. Moreover, it will facilitate the necessary process of amending the examples in situations when proposed notation is not machine-processable and DSL notation can not be inferred. The success of notation inference is directly influenced by number of language patterns that are to be looked for. Indeed, the ones listed in this paper certainly do not encompass all of the patterns that can be observed in computer languages. Since language patterns are still the subject of ongoing research, we expect to identify and formalize more of them in the future.

Acknowledgment

This work was supported by VEGA Grant No. 1/0305/11 Co-evolution of the artifacts written in domain-specific languages driven by language evolution.

References

[1] M. Črepinšek, M. Mernik, B. R. Bryant, F. Javed, and A. Sprague, "Inferring Context-Free Grammars for Domain-Specific Languages," *Electronic Notes in Theoretical Computer Science (ENTCS)*, vol. 141, pp. 99–116, December 2005. [Online]. Available: http://dx.doi.org/10.1016/j.entcs.2005.02.055

[2] P. Dupont, "Regular Grammatical Inference from Positive and Negative Samples by Genetic Search: the GIG Method," in *Proceedings of the Second International Colloquium on Grammatical Inference and Applications*. London, UK: Springer-Verlag, 1994, pp. 236–245. [Online]. Available: http://portal.acm.org/citation.cfm?id=645515.658234

[3] M. Mernik, D. Hrncic, B. Bryant, A. Sprague, J. Gray, Q. Liu, and F. Javed, "Grammar inference algorithms and applications in software engineering," in *Information, Communication and Automation Technologies, 2009. ICAT 2009. XXII International Symposium on*, 2009, pp. 1 –7.

[4] J. Porubän, M. Forgáč, M. Sabo, and M. Běhálek, "Annotation Based Parser Generator," *Computer Science an Information Systems*, vol. 7, no. 2, pp. 291–307, 2010.

[5] K. J. Lang, B. A. Pearlmutter, and R. A. Price, "Results of the Abbadingo One DFA Learning Competition and a New Evidence-Driven State Merging Algorithm," in *Proceedings of the 4th International Colloquium on Grammatical Inference*. London, UK: Springer-Verlag, 1998, pp. 1–12. [Online]. Available: http://portal.acm.org/citation.cfm?id=645517.655780

[6] J. Oncina and P. Garcia, "Inferring regular languages in polynomial update time," in *Pattern Recognition and Image Analysis*, 1991, pp. 49–61.

[7] M. Črepinšek, M. Mernik, and V. Žumer, "Extracting Grammar from Programs: Brute Force Approach," *SIGPLAN Not.*, vol. 40, pp. 29–38, April 2005. [Online]. Available: http://doi.acm.org/10.1145/1064165.1064171

[8] P. R. Henriques, M. J. V. Pereira, M. J. A. Var, and A. Pereira, "Automatic Generation of Language-based Tools," in *Electronic Notes in Theoretical Computer Science*, M. van den Brand and R. Laemmel, Eds., vol. 65, no. 3. Elsevier Science Publishers, 2002.

[9] R. Lämmel and C. Verhoef, "Semi-automatic Grammar Recovery," *Softw. Pract. Exper.*, vol. 31, pp. 1395–1448, December 2001. [Online]. Available: http://portal.acm.org/citation.cfm?id=569229.569230

[10] F. Javed, M. Mernik, J. Gray, and B. R. Bryant, "Mars: A metamodel recovery system using grammar inference," *Information and Software Technology*, vol. 50, pp. 948–968, August 2008. [Online]. Available: http://portal.acm.org/citation.cfm?id=1379905.1379993

[11] S. Cook, G. Jones, K. Stuart, and W. A. Cameron, *Domain-Specific Development with Visual Studio DSL Tools*. Addison-Wesley Professional, 2007.

[12] R. C. Gronback, *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*, 1st ed. Addison-Wesley Professional, 2009.

[13] T. Parr, *The Definitive Antlr Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007.

[14] EMFText, http://www.emftext.org.

[15] Xtext, http://www.eclipse.org/Xtext.

[16] D. Lakatoš, J. Porubän, and M. Sabo, "Assisted Software Language Creation using Internal Model," in *Proceedings of the International Conference on Engineering of Modern Electric Systems*, ser. EMES'11, 2011.

[17] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[18] J. Porubän, M. Sabo, J. Kollár, and M. Mernik, "Abstract Syntax Driven Language Development: Defining Language Semantics through Aspects," in *Proceedings of the International Workshop on Formalization of Modeling Languages*, ser. FML '10. New York, NY, USA: ACM, 2010, pp. 2:1–2:5.

[19] M. Sabo, "Abstract Syntax Driven Concrete Syntax Recognition," *Journal of Information, Control and Management Systems*, vol. 8, no. 4, pp. 393–402, 2010.

[20] M. Sabo and J. Porubän, "Concrete Syntax Recognition using Language Patterns," in *Proceedings of CSE 2010 International Scientific Conference on Computer Science and Engineering*, 2010, pp. 101–108.